



Controller-Service-Repository



Tom Collings · [Follow](#)

7 min read · Aug 10, 2021



Listen

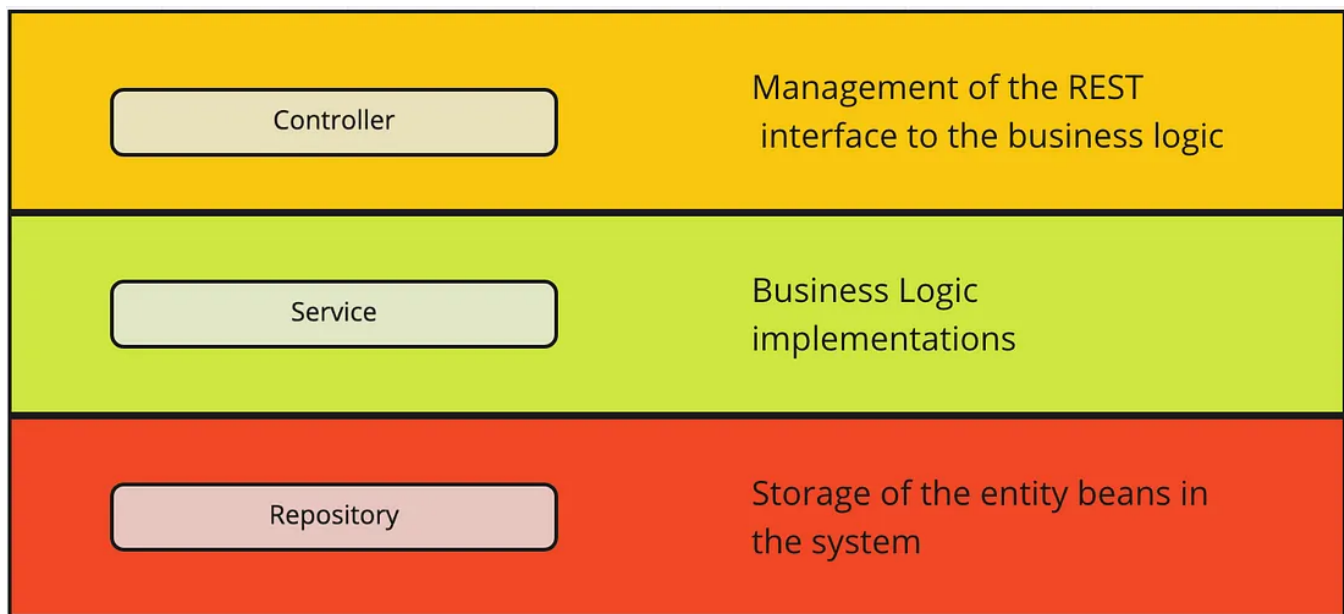


Share

I've had a unique opportunity fall into my lap. I've been spending a lot of my time over the past few years solving some of the stranger situations we find out in the field... little nuances that went undetected, that had huge unexpected consequences. Seemingly un-crack-able nuts that we found seams in to crack and unlock a bunch of new capabilities.

But this opportunity was something completely different. I'm working with folks who had just come out of coding bootcamps, and have no experience building software at all except for the small front-end samples they built as learning exercises.

It shouldn't have, but it surprised me to learn that while these new developers had seen the controller-service-repository pattern, they didn't understand any of the *why* behind that pattern. And they certainly hadn't seen anything around the testing of those layers and what made sense in that arena. So I gave an impromptu lecture on the topic, and the devs immediately asked if there were any written resources to support this talk. I responded that I will write one, so here we are.



The Controller-Service-Repository pattern is prevalent in a lot of Spring Boot applications. One of the big reasons I like this pattern is that it does a great job of a separation of concerns: The Controller layer, at the top of this picture, is solely responsible for exposing the functionality so that it can be consumed by external entities (including, perhaps, a UI component). The Repository layer, at the bottom of this picture, is responsible for storing and retrieving some set of data. The Service layer is where all the business logic should go. If the business logic requires fetching/saving data, it wires in a Repository. If someone wants to access this business logic, they go through a Controller to get there.

It's a pretty simple separation of concerns. If code is related to storage/retrieval, it should go in the Repository. If its dealing with exposing functionality, it goes in the Controller. Anything unique in the business logic would go in the Service layer. The Repository doesn't care which component is invoking it; it blindly does what it is asked. The Service layer doesn't care how it gets accessed, it just does its work, using a Repository where required. And the Controller is just passing the work down to the Service layer, so it can stay nice and lean.

Where this really begins to pay dividends is in the unit testing philosophies. By having a nice clean separation of concerns, we can mock adjacent layers and worry about only testing the concerns of that particular layer. Our Controller tests are only

worried about response codes and values, and we can mock the service to trigger those conditions. The Service layer can even be tested as a POJO, and by mocking Repository conditions we can test all the business logic therein without having to worry about going through the controller layer to test it.

Let's work through an example: Suppose we have an application that gets Cats in and out of a database. Our Entity object is very simple: Just the id and the name:

```
@Getter
@Setter
@With
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Cat {

    @Id
    private long id;

    private String name;

}
```

Here we're using the lombok annotations to keep our code cleaner. And as its an Entity object, a unit test doesn't make a lot of sense here.



Let's work top down: We want a controller method that finds a cat based on an id. Being good practitioners of TDD, we first write the test:

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(CatsController.class)
public class CatsControllerTest {
    @MockBean
    CatService catService;

    @Autowired
    MockMvc mockMvc;

    @Test
    public void testGetById() throws Exception {
        Cat cat = new Cat().withId(1).withName("Boots");

        when(catService.findById(1)).thenReturn(cat);

        ResultActions result = mockMvc.perform(get("/api/cat/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.name").value("Boots"));

        verify(catService).findById(1);
    }
}
```

The test is fairly straightforward. We're using Jupiter, so we can use the @ExtendWith annotations to make this a full Spring test that creates the Spring context. We also want to make this a MockMvcTest to give us the ability to mock our HTTP requests.

Because we want to **only** test the Controller layer in this unit test, we can mock the layer directly below it: The CatService layer. That's straightforward to mock, and we can set it up to return our cat Boots when the client asks for Cat with identifier 1. If the service returns that Cat object, then the controller should also return the same object with a 200 status.

The “verify” statement ensures that the Controller invokes the Service to get this value. After all, we could hard code the Controller layer to return Boots to get the test to pass, but that’s not going to work for every id in the long run.

So this test fails, because the controller isn’t doing what it should. So let’s build our Controller layer:

```
@RestController
public class CatsController {

    private CatService catService;

    public CatsController (CatService catService) {
        this.catService = catService;
    }

    @GetMapping("/api/cat/{id}")
    public Cat getCatById(@PathVariable long id) throws Exception {
        return catService.findById(id);
    }
}
```

Pretty straightforward, again. We have a single endpoint that returns whatever the Service provides. All this class does is expose some Service functionality RESTfully, which is exactly what our controller layer should do.

Now let’s build our service! Our test should ensure that the Service gets the requested Cat object from a repository and return it to whomever asked for it:

```
@ExtendWith(MockitoExtension.class)
public class CatServiceTest {

    @Mock
    CatRepository catRepo;

    @Test
```

```
public void getCatById () throws Exception {  
  
    CatService catService = new CatService(catRepo);  
  
    Optional<Cat> cat = Optional.of(new Cat().withId(1).withName("Boots"));  
  
    when(catRepo.findById(1L)).thenReturn(cat);  
  
    Cat foundCat = catService.findById(1);  
  
    assertThat(foundCat).isEqualTo(cat.get());  
  
    verify(catRepo).findById(1L);  
  
}  
}
```

The Service API is based on returning a Cat object, but from experience we know that the Spring JPA Repositories return Optional<Cat> from find operations. We'd like our Service to handle this Optional instead of returning to the client, so we Mock our Repository and build out method signatures accordingly. We also include the verify statement to ensure that the Repository layer is being invoked.

We should also point out that we're testing this Service class as a POJO: this means we're not instantiating the entire Spring Context just to test a simple bean. This means that it takes less time to run our tests.

Our test fails, as we expect, so let's build the Service class to get our test to pass:

```
@Service  
public class CatService {  
  
    private CatRepository catRepository;  
  
    public CatService(CatRepository catRepository) {  
        this.catRepository = catRepository;  
    }  
}
```

```
public Cat findById(long id) throws CatNotFoundException {  
  
    Optional<Cat> oCat = catRepository.findById(id);  
    return oCat.get();  
}  
}
```



And now we're cooking with Butter Flavored Crisco! the last thing to build is our Repository, but because Spring Data JPA builds the implementation for us, there's

not much value in creating tests at this layer:

```
public interface CatRepository extends CrudRepository<Cat, Long> {  
}
```

(Question for readers: When might it be useful to create a Repository layer test? There are several reasons we might want to consider this.)

Sharp eyed readers may have noticed something about this code: we're assuming that when we ask for a Cat, we're going to find one. As someone who has felines living in his house, I can assure you that this is not always the case. Cats have a great way of not being where they are supposed to be.



So how should we approach this? Traditionally, when we do a GET operation via id and that object is not found, we return a 404. Let's write up a test to get this condition via our service layer throwing a `CatNotFoundException`:

```
@Test  
public void testGetByIdNotFound() throws Exception {  
  
    when(catService.findById(1)).thenThrow(new CatNotFoundException());  
  
    ResultActions result = mockMvc.perform(get("/api/cat/1"))  
        .andExpect(status().isNotFound());  
  
    verify(catService).findById(1);  
}
```

And to get this to compile, we need to define our `CatNotFoundException`:

```
public class CatNotFoundException extends Exception {  
}
```

Because we have clean divisions between our layers, we can mock the behaviour at the Service layer and ensure that our Controller layer handles it well. We don't have to mock an entire repository with a missing Cat to make this happen.

I implemented this as a `ControllerAdvice` class, mostly because I like the idea of horizontal exception handling:

```
@ControllerAdvice  
public class ControllerAdvisor extends ResponseEntityExceptionHandler {  
  
    @ExceptionHandler(CatNotFoundException.class)  
    public ResponseEntity<Cat> handleCatNotFound(CatNotFoundException exc,  
        WebRequest req) {  
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);  
    }  
  
}
```

Without getting into the details too much, a class with this annotation will be able to handle exceptions across **all** Controller actions in the application. So if something in the future decides to throw a `CatNotFoundException`, we've already handled that exception in this class.

So we fixed the Controller layer, now we need to write the business code at the Service layer. First, the test:

```
@Test  
public void getCatByIdNotFound() {  
    CatService catService = new CatService(catRepo);  
  
    Optional<Cat> cat = Optional.empty();
```

```
when(catRepo.findById(1L)).thenReturn(cat);

assertThrows(CatNotFoundException.class, () -> {
    Cat foundCat = catService.findById(1);
});

verify(catRepo).findById(1L);
}
```

Again, our separation of concerns is paying dividends. We can mock the Repository behaviour to return an empty `Optional<Cat>` to force our Service layer condition. We can test this case with some very simple code that doesn't have to run through a Controller layer or work with an instantiated Repository.

Let's get the test to pass:

```
public Cat findById(long id) throws CatNotFoundException {

    Optional<Cat> oCat = catRepository.findById(id);

    if (oCat.isEmpty()) {
        throw new CatNotFoundException();
    }

    return oCat.get();
}
```

We added the business-specific logic of a Cat not found to the Service layer. We added a specific response code based on that condition to the Controller layer. And the tests to validate these conditions were simple to write because we could mock the necessary conditions on the adjacent layers.

As you're building your back end services in Spring Boot, this Controller-Service-Repository pattern can pay a lot of dividends. It keeps your code clean, it keeps your tests simple, and it makes it clear where new code should go.

Code for this sample can be found at:

<https://github.com/tom-collings/cats>